Hydra HeadV1 Specification: Coordinated Head protocol

DRAFT

Sebastian Nagel sebastian.nagel@iohk.io

November 22, 2023

1 Introduction

This document specifies the 'Coordinated Hydra Head' protocol to be implemented as the first version of Hydra Head on Cardano - **Hydra HeadV1**. The protocol is derived from variants described in the original paper [8], but was further simplified to make a first implementation on Cardano possible.

Note that the format and scope of this document is (currently) also inspired by the paper and hence does not include a definition of the networking protocols or concrete message formats. It is structured similarly, but focuses on a single variant, and avoids indirections and unnecessary generalizations. The document is kept in sync with the reference implementation available on Github [4]. Red sections indicate that they are currently not covered or missing in the implementation, where blue parts mean a difference in how it is realized.

Add: network specification (message formats)

First, a high-level overview of the protocol and how it differs from legacy variants of the Head protocol is given in Section 2. Relevant definitions and notations are introduced in Section 3, while Section 4 describes protocol setup and assumptions. Then, the actual on-chain transactions of the protocol are defined in Section 5, before the off-chain protocol part specifies behavior of Hydra parties off-chain and ties the knot with on-chain transactions in Section 6. At last, Section 7 gives the security definition, properties and proofs for the Coordinated Head protocol.

2 Protocol Overview

The Hydra Head protocol provides functionality to lock a set of UTxOs on a blockchain, referred to as the *mainchain*, and evolve it inside a so-called off-chain *head*, independently of the mainchain. At any point, the head can be closed with the effect that the locked set of UTxOs on the mainchain is replaced by the latest set of UTxOs inside the head. The protocol guarantees full wealth preservation: no generation of funds can happen off-chain (inside a head) and no responsive honest party involved in a head can ever lose any funds other than by consenting to give them away. In exchange for decreased liveness guarantees (stop any time), it can essentially proceed at network speed under good conditions, thereby reducing latency and increasing throughput. At the same time, the head protocol provides the same capabilities as the mainchain by reusing the same ledger model and transaction formats — making the protocol "isomorphic".

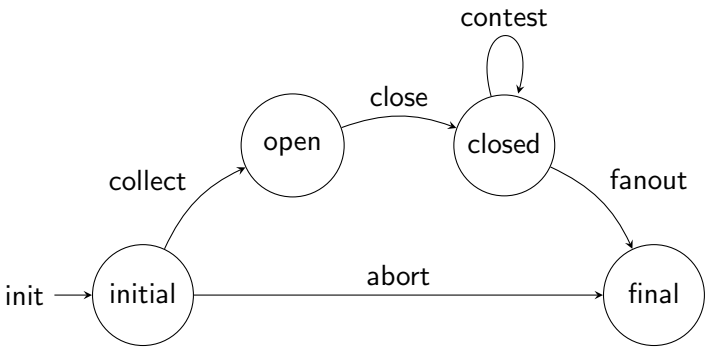

Figure 1: Mainchain state diagram for this version of the Hydra protocol.

2.1 Opening the head

To create a head-protocol instance, any party may take the role of an *initiator* and ask other parties, the *head members*, to participate in the head by exchanging public keys and agreeing on other protocol parameters. These public keys are used for both, the authentication of head-related on-chain transactions that are restricted to head members (e.g., a non-member is not allowed to close the head) and for signing off-chain transactions in the head.

The initiator then establishes the head by submitting an *initial* transaction to the mainchain that contains the Hydra protocol parameters and mints special *participation tokens (PT)* identifying the head members. The *initial* transaction also initializes a state machine (see Fig. 1) that manages the "transfer" of UTxOs into the head and back. The state machine comprises the four states: initial, open, closed, and final. A *state thread token (ST)* minted in *initial* marks the head output and ensures contract continuity [6].

Once the initial transaction appears on the mainchain, establishing the initial state initial, each head member can attach a *commit* transaction, which locks (on the mainchain) the UTxOs that the party wants to commit to the head, or deliberately acknowledges to commit nothing.

The commit transactions are subsequently collected by the *collectCom* transaction causing a transition from initial to open. Once the open state is confirmed, the head members start running the off-chain head protocol, which evolves the initial UTxO set (the union over all UTxOs committed by all head members) independently of the mainchain. For the case where some head members fail to post a *commit* transaction, the head can be aborted by going directly from initial to final.

2.2 The Coordinated Head protocol

The actual Head protocol starts after the initialization phase with an initial set of UTxOs that is identical to the UTxOs locked on-chain via the *commit* and *collectCom* transactions.

The protocol processes off-chain transactions by distributing them between participants, while each party maintains their view of the local UTxO state. That is, the current set of UTxOs evolved from the initial UTxO set by applying transactions as they are received from the other parties.

To confirm transactions and allow for an on-chain decommit of the resulting UTxO set without needing the whole transaction history, snapshots are created by the protocol participants. The initial snapshot U_0 corresponds to the initial UTxO set, while snapshots thereafter $U_1, U_2, \ldots$ are created with monotonically increasing snapshot numbers.

For this, the next snapshot leader (round-robin) requests his view of a new confirmed state to be signed by all participants as a new snapshot. The leader does not need to send his local state, but only indicate, by hashes, the set of transactions to be included in order to obtain the to-be-snapshotted UTxO set.

The other participants sign the snapshot as soon as they have (also) seen the transactions that are to be processed on top of its preceding snapshot: a party's local state is always ahead of the latest confirmed snapshot.

Signatures are broadcast and aggregated by each party. When all signature parts of the multi-signature are received and verified, a snapshot is considered confirmed. As a consequence, a participant can safely delete (if wished) all transactions that have been processed into it as the snapshot's multi-signature is now evidence that this state once existed during the head evolution.

2.3 Closing the head

The head protocol is designed to allow any head member at any point in time to produce, without interaction, a certificate to close the head. This certificate is created from the latest confirmed snapshot, specifically from its snapshot number and the respective multisignature. Using this certificate, the head member may "force close" the head by advancing the mainchain state machine to the closed state.

Once in closed, the state machine grants parties a contestation period, during which parties may contest the closure by posting the certificate of a newer snapshot on-chain in a contest transaction. Contesting leads back to the state closed and each party can contest at most once. After the contestation period has elapsed, the state machine may proceed to the final state. The state machine enforces that the outputs of the transaction leading to final correspond exactly to the latest UTxO set seen during the contestation period.

2.4 Differences

In the Coordinated Head protocol, off-chain consensus is simplified by not having transactions confirmed concurrently to the snapshots (and to each other) but having the snapshot leader propose, in their snapshot, a set of transactions for explicit confirmation. The parties' views of confirmed transactions thus progress in sync with each other (once per confirmed snapshot), thus simplifying the close/contest procedure on the mainchain. Also, there is no need for conflict resolution as in Appendix B of [8]. In summary, the differences to the original Head protocol in [8] are:

- No hanging transactions due to 'coordination'.
- No acknowledgement nor confirmation of transactions.
- No confirmation message for snapshots (two-round local confirmation).

3 Preliminaries

This section introduces notation and other preliminaries used in the remainder of the specification.

3.1 Notation

The specification uses set-notation based approach while also inspired by [1] and [6]. Values a are in a set $a \in \mathcal{A}$, also indicated as being of some type $a : \mathcal{A}$, and multidimensional values are tuples drawn from a $\times$ product of multiple sets, e.g. $(a, b) \in (\mathcal{A} \times \mathcal{B})$. An empty set is indicated by $\emptyset$ and sets may be enumerated using $\{a_1 \dots a_n\}$ notation. The $=$ operator means equality and $\leftarrow$ is explicit assignment of a variable or value to one or more variables. Projection is used to access the elements of a tuple, e.g. $(a, b)^{\downarrow 1} = a$. Functions are morphisms mapping from one set to another $x : \mathcal{A} \to f(x) : \mathcal{B}$, where function application of a function f to an argument x is written as $f(x)$.

Furthermore, given a set $\mathcal{A}$, let

- $\mathcal{A}^? = \mathcal{A} \cup \Diamond$ denotes an option: a value from $\mathcal{A}$ or no value at all,
- $\mathcal{A}^n$ be the set of all n-sized sequences over $\mathcal{A}$,
- $\mathcal{A}^! = \bigcup_{i=1}^{n \in \mathbb{N}} \mathcal{A}^i$ be the set of non-empty sequences over $\mathcal{A}$, and
- $\mathcal{A}^* = \bigcup_{i=0}^{n \in \mathbb{N}} \mathcal{A}^i$ be the set of all sequences over $\mathcal{A}$.

With this, we further define:

- $\mathbb{B} = \{\texttt{false}, \texttt{true}\}$ are boolean values
- $\mathbb{N}$ are natural numbers $\{0, 1, 2, \dots\}$
- $\mathbb{Z}$ are integer numbers $\{\dots, -2, -1, 0, 1, 2, \dots\}$
- $\mathbb{H} = \bigcup_{n=0}^{\inf} \{0, 1\}^{8n}$ denotes a arbitrary string of bytes
- $\mathsf{concat} : \mathbb{H}^* \to \mathbb{H}$ is concatenating bytes, we also use operator $\bigoplus$ for this
- $\mathsf{hash} : x \to \mathbb{H}$ denotes a collision-resistant hashing function and $x^{\#}$ indicates the hash of x
- $\mathsf{bytes} : x \to \mathbb{H}$ denotes an invertible serialisation function mapping arbitrary data to bytes
- $a||b = \mathsf{concat}(\mathsf{bytes}(a), \mathsf{bytes}(b))$ is an operator which concatenates the $\mathsf{bytes}(b)$ to the $\mathsf{bytes}(a)$
- Lists of values $l \in \mathcal{A}^*$ are written as $l = [x_1, \dots, x_n]$. Empty lists are denoted by [], the ith element x_i is also written $l[i]$ and the length of the list is $|l| = n$. An underscore is also used to indicate a list of values $\underline{x} = l$. Projection on lists are mapped to their elements, i.e. $\underline{x}^{\downarrow 1} = [x_1^{\downarrow 1}, \dots, x_n^{\downarrow 1}]$.
- $\mathsf{sortOn} : i \to \mathcal{A}^* \to \mathcal{A}^*$ does sort a list of values on the ith projection.
- Data is a universal data type of nested sums and products built up recursively from the base types of $\mathbb{Z}$ and $\mathbb{H}$.

3.2 Public key multi-signature scheme

A multisignature scheme is a set of algorithms where

- MS-Setup generates public parameters Π, such that
- $(k^{ver}, k^{sig}) \leftarrow \mathsf{MS\text{-}KG}(\Pi)$ can be used to generate fresh key pairs,
- $\sigma \leftarrow \mathsf{MS\text{-}Sign}(\Pi, k^{sig}, m)$ signs a message m using key k^{sig},
- $\tilde{k} \leftarrow \mathsf{MS\text{-}AVK}(\Pi, \overline{k})$ aggregates a list of verification keys $\overline{k}$ into a single, aggregate key $\tilde{k}$,
- $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\Pi, m, \overline{k}, \overline{\sigma})$ aggregates a list of signatures $\overline{\sigma}$ about message m into a single, aggregate signature $\tilde{\sigma}$.
- $\mathsf{MS\text{-}Verify}(\Pi, \tilde{k}, m, \tilde{\sigma}) \in \mathbb{B}$ verifies an aggregate signature $\tilde{\sigma}$ of message m under an aggregate verification key $\tilde{k}$.

The security definition of a multisignature scheme from [9, 10] guarantees that, if $\tilde{k}$ is produced from a tuple of verification keys $\overline{k}$ via MS-AVK, then no aggregate signature $\tilde{\sigma}$ can pass verification $\mathsf{MS\text{-}Verify}(\tilde{k}, m, \tilde{\sigma})$ unless all honest parties holding keys in $\overline{k}$ signed m.

Note that in the following, we make the parameter Π implicit and leave out the *ver* suffix for verification key such that $k = k^{ver}$ for better readability.

3.3 Extended UTxO

The Hydra Head protocol is specified to work on the so-called Extended UTxO (EUTxO) ledgers like Cardano.

The basis for EUTxO is Bitcoin's UTxO ledger model [5, 11]. Intuitively, it arranges transactions in a directed acyclic graph, such as the one in Figure 2, where boxes represent transactions with (red) inputs to the left and (black) outputs to the right. A dangling (unconnected) output is an *unspent transaction output (UTxO)* — there are two UTxOs in the figure.

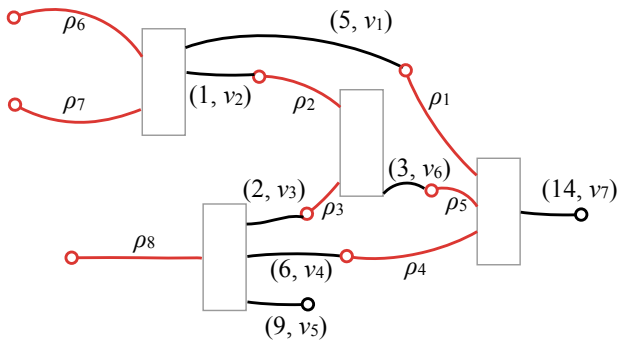

Figure 2: Example of a plain UTxO graph

The following paragraphs will give definitions of the UTxO model and it's extension to support scripting (EUTxO) suitable for this Hydra Head protocol specification. For a more detailed introduction to the EUTxO ledger model, see [6], [1] and [7].

3.3.1 Values

Definition 1 (Values). *Values are sets that keep track of how many units of which tokens of which currency are available. Given a finitely supported function $\mapsto$, that maps keys to monoids, a value is the set of such mappings over currencies (minting policy identifiers), over a mapping of token names t to quantities q:*

$$\mathsf{val} \in \mathsf{Val} = (c : \mathbb{H} \mapsto (t : \mathbb{H} \mapsto q : \mathbb{Z}))$$

where addition of values is defined as $+$ and $\varnothing$ is the empty value.

For example, the value $\{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\}$ contains tokens t_1 and t_2 of currency c_1 and addition merges currencies and token names naturally:

$$\begin{aligned} & \{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\} \\ + & \{c_1 \mapsto \{t_2 \mapsto 1, t_3 \mapsto 1\}, c_2 \mapsto \{t_1 \mapsto 2\}\} \\ = & \{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 2, t_3 \mapsto 1\}, c_2 \mapsto \{t_1 \mapsto 2\}\} \ . \end{aligned}$$

While the above definition should be sufficient for the purpose of this specification, a full definition for finitely supported functions and values as used here can be found in [7]. To further improve readability, we define the following shorthands:

- $\{t_1, \ldots, t_n\} :: c$ for a set positive single quantity assets $\{c \mapsto \{t_1 \mapsto 1, \ldots, t_n \mapsto 1\}\}$,
- $\{t_1, \ldots, t_n\}^{-1} :: c$ for a set of negative single quantity assets $\{c \mapsto \{t_1 \mapsto -1, \ldots, t_n \mapsto -1\}\}$,
- $\{c \mapsto t \mapsto q\}$ for the value entry $\{c \mapsto \{t \mapsto q\}\}$,
- $\{c \mapsto \cdot \mapsto q\}$ for any asset with currency c and quantity q irrespective of token name.

3.3.2 Scripts

Validator scripts are called *phase-2* scripts in the Cardano Ledger specification (see [3] for a formal treatment of these). Scripts are used for multiple purposes, but most often (and sufficient for this specification) as a *spending* or *minting* policy script.

Definition 2 (Minting Policy Script). *A script $\mu \in \mathcal{M}$ governing whether a value can be minted (or burned), is a pure function with type*

$$\mu \in \mathcal{M} = (\rho : \mathsf{Data}) \to (\gamma : \Gamma) \to \mathbb{B},$$

where $\rho \in \mathsf{Data}$ is the redeemer provided as part of the transaction being validated and $\gamma \in \Gamma$ is the validation context.

Definition 3 (Spending Validator Script). *A validator script $\nu \in \mathcal{V}$ governing whether an output can be spent, is a pure function with type*

$$\nu \in \mathcal{V} = (\delta : \mathsf{Data}) \to (\rho : \mathsf{Data}) \to (\gamma : \Gamma) \to \mathbb{B},$$

where $\delta \in \mathsf{Data}$ is the datum available at the output to be spent, $\rho \in \mathsf{Data}$ is the redeemer data provided as part of the transaction being validated, and $\gamma \in \Gamma$ is the validation context.

3.3.3 Transactions

We define EUTxO inputs, outputs and transactions as they are available to scripts and just enough to specify the behavior of the Hydra validator scripts. For example outputs addresses and datums are much more complicated in the full ledger model [1, 2].

Definition 4 (Outputs). *An output $o \in \mathcal{O}$ stores some value $\mathsf{val} \in \mathsf{Val}$ at some address, defined by the hash of a validator script $\nu^{\#} \in \mathbb{H} = \mathsf{hash}(\nu \in \mathcal{V})$, and may store (reference) some data $\delta \in \mathsf{Data}$:*

$$o \in \mathcal{O} = (\mathsf{val} : \mathsf{Val} \times \nu^{\#} : \mathbb{H} \times \delta : \mathsf{Data})$$

Definition 5 (Output references). *An output reference $\phi \in \Phi$ points to an output of a transaction, using a transaction id (that is, a hash of the transaction body) and the output index within that transaction.*

$$\phi \in \Phi = (\mathbb{H} \times \mathbb{N})$$

Definition 6 (Inputs). *A transaction input $i \in \mathcal{I}$ is an output reference $\phi \in \Phi$ with a corresponding redeemer $\rho \in \mathsf{Data}$:*

$$i \in \mathcal{I} = (\phi : \Phi \times \rho : \mathsf{Data})$$

Definition 7 (Validation Context). *A validation context $\gamma \in \Gamma$ is a view on the transaction to be validated:*

$$\gamma \in \Gamma = (\mathcal{I}^* \times \mathcal{O}^* \times \mathsf{Val} \times \mathcal{S}^{\leftrightarrow} \times \mathcal{K})$$

*where $\mathcal{I} \in \mathcal{I}^*$ is a **set** of inputs, $\mathcal{O} \in \mathcal{O}^*$ is a **list** of outputs, $\mathsf{mint} \in \mathsf{Val}$ is the minted (or burned) value, $(t_{\mathsf{min}}, t_{\mathsf{max}}) \in \mathcal{S}^{\leftrightarrow}$ are the lower and upper validity bounds where $t_{\mathsf{min}} <= t_{\mathsf{max}}$, and $\kappa \in \mathcal{K}$ is the set of verification keys which signed the transaction.*

Informally, scripts are evaluated by the ledger when it applies a transaction to its current state to yield a new ledger state (besides checking the transaction integrity, signatures and ledger rules). Each validator script referenced by an output is passed its arguments drawn from the output it locks and the transaction context it is executed in. The transaction is valid if and only if all scripts validate, i.e. $\mu(\rho, \gamma) = \texttt{true}$ and $\nu(\delta, \rho, \gamma) = \texttt{true}$.

3.3.4 State machines and graphical notation

State machines in the EUTxO ledger model are commonly described using the *constraint emitting machine (CEM)* formalism [6], e.g. the original paper describes the Hydra Head protocol using this notation [8]. Although inspired by CEMs, this specification uses a more direct representation of individual transactions to simplify description of non-state-machine transactions and help translation to concrete implementations on Cardano. The structure of the state machine is enforced on-chain through *scripts* which run as part of the ledger's validation of a transaction (see Section 3.3). For each protocol transaction, the specification defines the structure of the transaction and enumerates the transaction constraints enforced by the scripts ($\text{tx}^{\equiv}$ in the CEM formalism).

Add an example graph with a legend

4 Protocol Setup

In order to create a head-protocol instance, an initiator invites a set of participants (the initiator being one of them) to join by announcing to them the protocol parameters.

- For on-chain transaction authentication (Cardano) purposes, each party p_i generates a corresponding key pair (k_i^{ver}, k_i^{sig}) and sends their verification key k_i^{ver} to all other parties. In the case of Cardano, these are Ed25519 keys.
- For off-chain signing (Hydra) purposes, a very basic multisignature scheme (MS, as defined in Section 3.2) based on EdDSA using Ed25519 keys is used:
  - MS-KG is Ed25519 key generation (requires no parameters)
  - MS-Sign creates an EdDSA signature
  - MS-AVK is concatenation of verification keys into an ordered list
  - MS-ASig is concatenation of signatures into an ordered list
  - MS-Verify verifies the "aggregate" signature by verifying each individual EdDSA signature under the corresponding Ed25519 verification key

  To help distinguish on- and off-chain key sets, Cardano verification keys are written k_{C}, while Hydra verification keys are indicated as k_{H} for the remainder of this document.
- Each party p_i generates a hydra key pair and sends their hydra verification key to all other parties.
- Each party p_i computes the aggregate key from the received verification keys, stores the aggregate key, their signing key as well as the number of participants n.
- Each party establishes pairwise communication channels to all other parties. That is, every network message received from a specific party is checked for (channel) authentication. It is the implementer's duty to find a suitable authentication process for the communication channels.
- All parties agree on a contestation period T.

If any of the above fails (or the party does not agree to join the head in the first place), the party aborts the initiation protocol and ignores any further action. Finally, at least one of the participants posts the *init* transaction onchain as described next in Section 5.

5 On-chain Protocol

The following sections describe the the *on-chain* protocol controlling the life-cycle of a Hydra head, which can be intuitively described as a state machine (see Figure 1). Each transition in this state machine is represented and caused by a corresponding Hydra protocol transaction on-chain: *init*, *commit*, *abort*, *collectCom*, *close*, *contest*, and *fanout*.
The protocol defines one minting policy script and three validator scripts:

- μ_{head} governs minting of state and participation tokens in *init* and burning of these tokens in *abort* and *fanout*.
- ν_{initial} controls how UTxOs are committed to the head in *commit* or when the head initialization is aborted via *abort*.
- ν_{commit} controls the collection of committed UTxOs into the head in *collectCom* or that funds are reimbursed in an *abort*.
- ν_{head} represents the main protocol state machine logic and ensures contract continuity throughout *collectCom*, *close*, *contest* and *fanout*.

Open problem: ensure abort is always possible. e.g. by individual aborts or undoing commits

Open problem: ensure fanout is always possible, e.g. by limiting complexity of U_0

5.1 Init transaction

The *init* transaction creates a head instance and establishes the initial state of the protocol and is shown in Figure 3. The head instance is represented by the unique currency identifier cid created by minting tokens using the μ_{head} minting policy script which is parameterized by a single output reference parameter $\phi_{\mathsf{seed}} \in \Phi$:

$$\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\phi_{\mathsf{seed}}))$$

Two kinds of tokens are minted:

- A single *State Thread (ST)* token marking the head output. This output contains the state of the protocol on-chain and the token ensures contract continuity. The token name is the well known string `HydraHeadV1`, i.e. $\mathsf{ST} = \{\mathsf{cid} \mapsto \texttt{HydraHeadV1} \mapsto 1\}$.
- One *Participation Token (PT)* per participant $i \in \{1 \dots n\}$ to be used for authenticating further transactions and to ensure every participant can commit and cannot be censored. The token name is the participant's verification key hash $k_i^{\#} = \mathsf{hash}(k_i^{ver})$ of the verification key as received during protocol setup, i.e. $\mathsf{PT}_i = \{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\}$.

Consequently, the *init* transaction

- has at least input ϕ_{seed},
- mints the state thread token ST, and one PT for each of the n participants with policy cid,
- has n initial outputs o_{initial_i} with datum $\delta_{\mathsf{initial}} = \mathsf{cid}$,
- has one head output o_{head}, which captures the initial state of the protocol in the datum

  $$\delta_{\mathsf{head}} = (\mathsf{initial}, \mathsf{cid}', \phi'_{\mathsf{seed}}, \tilde{k}_{\mathsf{H}}, n, T)$$

  where

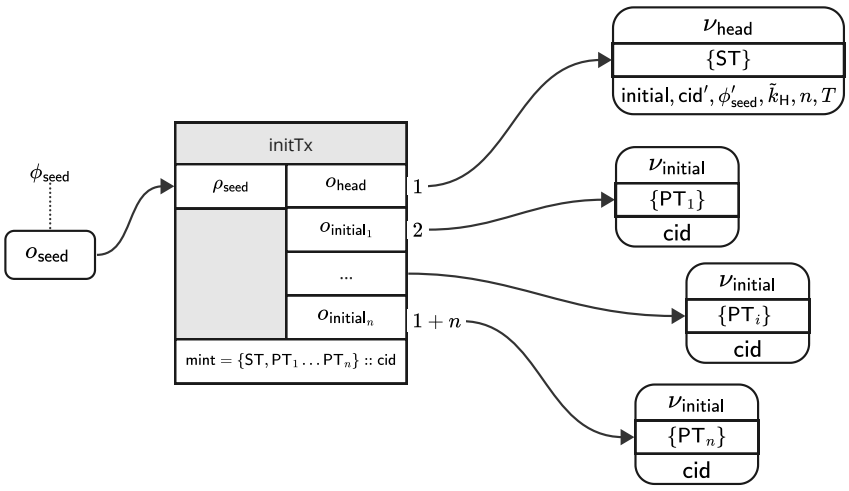

Figure 3: *init* transaction spending a seed UTxO, and producing the head output in state initial and initial outputs for each participant.

- initial is a state identifier,
- cid′ is the unique currency id of this instance,
- ϕ'_{seed} is the output reference parameter of μ_{head},
- $\tilde{k}_{\text{H}}$ is the aggregated off-chain multi-signature key established during the setup phase,
- n is the number of head participants, and
- T is the contestation period.

The $\mu_{\text{head}}(\phi_{\text{seed}})$ minting policy is the only script that verifies init transactions and can be redeemed with either a mint or burn redeemer:

- When evaluated with the mint redeemer,
  1. The seed output is spent in this transaction. This guarantees uniqueness of the policy cid because the EUTxO ledger ensures that ϕ_{seed} cannot be spent twice in the same chain. $(\phi_{\text{seed}}, \cdot) \in \mathcal{I}$
  2. All entries of mint are of this policy and of single quantity $\forall \{c \mapsto \cdot \mapsto q\} \in \text{mint} : c = \text{cid} \wedge q = 1$
  3. Right number of tokens are minted $|\text{mint}| = n + 1$
  4. State token is sent to the head validator $\text{ST} \in \text{val}_{\text{head}}$
  5. The correct number of initial outputs are present $|(\cdot, \nu_{\text{initial}}, \cdot) \in \mathcal{O}| = n$
  6. All participation tokens are sent to the initial validator as an initial output $\forall i \in [1 \dots n] : \{\text{cid} \mapsto \cdot \mapsto 1\} \in \text{val}_{\text{initial}_i}$
  7. The δ_{head} contains own currency id $\text{cid} = \text{cid}'$ and the right seed reference $\phi_{\text{seed}} = \phi'_{\text{seed}}$
  8. All initial outputs have a cid as their datum: $\forall i \in [1 \dots n] : \text{cid} = \delta_{\text{initial}_i}$

- When evaluated with the burn redeemer,
  1. All tokens for this policy in mint need to be of negative quantity $\forall\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint} : q < 0$.

Important: The μ_{head} minting policy only ensures uniqueness of cid, that the right amount of tokens have been minted and sent to ν_{head} and ν_{initial} respectively, while these validators in turn ensure continuity of the contract. However, it is **crucial** that all head members check that head output always contains an ST token of policy cid which satisfies $\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\phi_{\mathsf{seed}}))$. The ϕ_{seed} from a head datum can be used to determine this. Also, head members should verify whether the correct verification key hashes are used in the PTs and the initial state is consistent with parameters agreed during setup. See the initialTx behavior in Figure 10 for details about these checks.

5.2 Commit Transaction

A *commit* transaction may be submitted by each participant $\forall i \in \{1 \dots n\}$ to commit some UTxO into the head or acknowledge to not commit anything. The transaction is depicted in Figure 4 and has the following structure:

- One input spending from ν_{initial} with datum $\delta_{\mathsf{initial}}$, where value $\mathsf{val}_{\mathsf{initial}_i}$ holds a PT_i, and the redeemer $\rho_{\mathsf{initial}} \in \Phi^?$ is an optional output reference to be committed,
- zero or more inputs with reference $\phi_{\mathsf{committed}_j}$ spending output $o_{\mathsf{committed}_j}$ with $\mathsf{val}_{\mathsf{committed}_j}$,
- one output paying to ν_{commit} with value $\mathsf{val}_{\mathsf{commit}_i}$ and datum δ_{commit}.

The ν_{initial} validator with $\delta_{\mathsf{initial}} = \mathsf{cid}$ and $\rho_{\mathsf{initial}} = \underline{\phi}_{\mathsf{committed}}$ ensures that:

1. All committed value is in the output $\mathsf{val}_{\mathsf{commit}_i} \supseteq \mathsf{val}_{\mathsf{initial}_i} \cup (\bigcup_{j=1}^{m} \mathsf{val}_{\mathsf{committed}_j})$ [1]
2. Currency id and committed outputs are recorded in the output datum $\delta_{\mathsf{commit}} = (\mathsf{cid}, C_i)$, where $C_i = \forall j \in \{1 \dots m\} : [(\phi_{\mathsf{committed}_j}, \mathsf{bytes}(o_{\mathsf{committed}_j}))]$ is a list of all committed UTxO recorded as tuples on-chain.
3. Transaction is signed by the right participant $\exists\{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\} \in \mathsf{val}_{\mathsf{initial}} \Rightarrow k_i^{\#} \in \kappa$
4. No minting or burning $\mathsf{mint} = \varnothing$

The ν_{commit} validator ensures the output is collected by either a *collectCom* in Section 5.3 or *abort* in Section 5.4 transaction of the on-chain state machine, selected by the appropriate redeemer.

update with multiple commits

5.3 CollectCom Transaction

The *collectCom* transaction (Figure 5) collects all the committed UTxOs to the same head. It has

- one input spending from ν_{head} holding the ST with δ_{head},

[1] The $\supseteq$ is important for real world situations where the values might not be exactly equal due to ledger constraints (i.e. to ensure a minimum value on outputs).

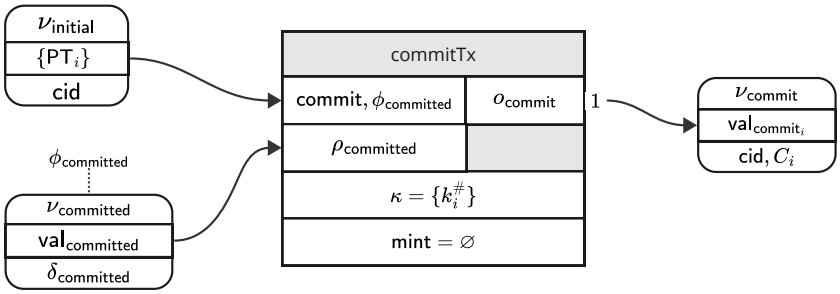

Figure 4: *commit* transaction spending an initial output and a single committed output, and producing a commit output.

- $\forall i \in \{1 \dots n\}$ inputs spending commit outputs $(\text{val}_{\text{commit}_i}, \nu_{\text{commit}}, \delta_{\text{commit}_i})$ with $\text{PT}_i \in \text{val}_{\text{commit}_i}$ and $\delta_{\text{commit}_i} = (\text{cid}, C_i)$, and
- one output paying to ν_{head} with value $\text{val}'_{\text{head}}$ and datum δ'_{head}.

The state-machine validator ν_{head} is spent with $\rho_{\text{head}} = \text{collect}$ and checks:

1. State is advanced from $\delta_{\text{head}} \sim \text{initial}$ to $\delta'_{\text{head}} \sim \text{open}$, parameters $\text{cid}, \tilde{k}_{\text{H}}, n, T$ stay unchanged and the new state is governed again by ν_{head}:

$$(\text{initial}, \text{cid}, \phi_{\text{seed}}, \tilde{k}_{\text{H}}, n, T) \xrightarrow{\text{collect}} (\text{open}, \text{cid}, \tilde{k}_{\text{H}}, n, T, \eta)$$

2. Commits collected in $\eta = (0, U^{\#})$ with snapshot number 0, where

$$U^{\#} = \text{combine}([C_1, \dots, C_n])$$

$$\text{combine}(\underline{C}) = \text{hash}(\text{concat}(\text{sortOn}(1, \text{concat}(\underline{C}))^{\downarrow 2}))$$

   That is, given a list of committed UTxO $\underline{C}$, where each element is a list of output references and the serialised representation of what was committed, combine first concatenates all commits together, sorts this list by the output references, concatenates all bytes and hashes the result[2].

3. All committed value captured and no value is extracted $\text{val}'_{\text{head}} = \text{val}_{\text{head}} \cup (\bigcup_{i=1}^{n} \text{val}_{\text{commit}_i})$.
4. Every participant had the chance to commit, by checking all tokens are present in output[3] $|\{\text{cid} \to . \to 1\} \in \text{val}'_{\text{head}}| = n + 1$.
5. Transaction is signed by a participant $\exists\{\text{cid} \mapsto k_i^{\#} \mapsto 1\} \in \text{val}_{\text{commit}_i} \Rightarrow k_i^{\#} \in \kappa$.
6. No minting or burning $\text{mint} = \varnothing$.

Each spent ν_{commit} validator with $\delta_{\text{commit}_i} = (\text{cid}, \cdot)$ and $\rho_{\text{commit}_i} = \text{collect}$ ensures that:

1. The state token of currency cid is present in the output value $\text{ST} \in \text{val}'_{\text{head}}$.

[2]Sorting is required to ensure a canonical representation which can also be reproduced from the UTxO set later in the fanout.

[3]This is sufficient as a Head participant would check off-chain whether a Head is initialized correctly with the right number of tokens.

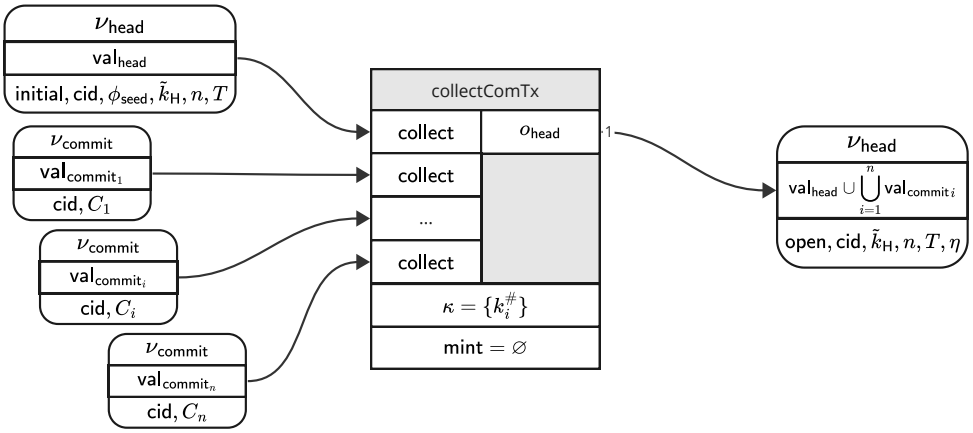

Figure 5: *collectCom* transaction spending the head output in initial state and collecting from multiple commit outputs into a single open head output.

5.4 Abort Transaction

The *abort* transaction (see Figure 6) allows a party to abort the creation of a head and consists of

- one input spending from ν_{head} holding the ST with δ_{head},
- $\forall i \in \{1 \dots n\}$ inputs either
  - spending from ν_{initial} with with $\mathsf{PT}_i \in \mathsf{val}_{\mathsf{initial}_i}$ and $\delta_{\mathsf{initial}_i} = \mathsf{cid}$, or
  - spending from ν_{commit} with with $\mathsf{PT}_i \in \mathsf{val}_{\mathsf{commit}_i}$ and $\delta_{\mathsf{commit}_i} = (\mathsf{cid}, C_i)$,
- outputs $o_1 \dots o_m$ to redistribute already committed UTxOs.

Note that *abort* represents a final transition of the state machine and hence there is no state machine output.

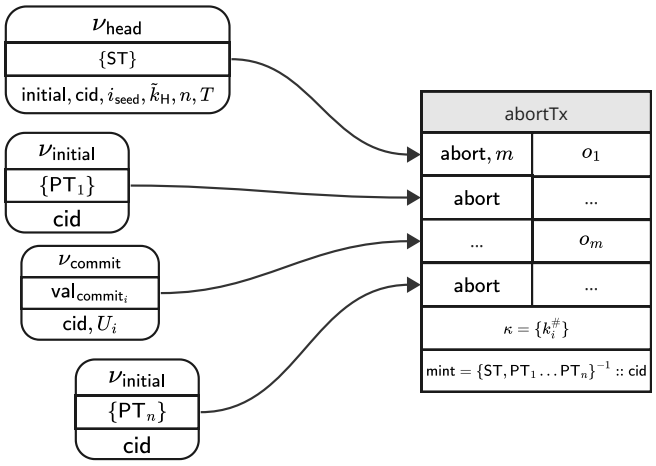

Figure 6: *abort* transaction spending the initial state head output and collecting all initial and commit outputs, which get reimbursed by outputs $o_1 \ldots o_m$. Note that each PT may be in either, an initial or commit output.

The state-machine validator ν_{head} is spent with $\rho_{\text{head}} = (\text{abort}, m)$, where m is the number of outputs to reimburse, and checks:

1. State is advanced from $\delta_{\text{head}} \sim \text{initial}$ to terminal state final:

$$(\text{initial}, \text{cid}, \phi_{\text{seed}}, \tilde{k}_{\text{H}}, n, T) \xrightarrow[m]{\text{abort}} \text{final}.$$

2. All UTxOs committed into the head are reimbursed exactly as they were committed. This is done by comparing hashes of serialised representations of the m reimbursing outputs $o_1 \ldots o_m$[4] with the canonically combined committed UTxOs in C_i:

$$\text{hash}(\bigoplus_{j=1}^{m} \text{bytes}(o_j)) = \text{combine}([C_i \mid \forall [1 \ldots n], C_i \neq \bot])$$

3. Transaction is signed by a participant $\exists \{\text{cid} \mapsto k_i^{\#} \mapsto -1\} \in \text{mint} \Rightarrow k_i^{\#} \in \kappa$.
4. All tokens are burnt $|\{\text{cid} \mapsto \cdot \mapsto -1\} \in \text{mint}| = n + 1$.

Each spent ν_{initial} validator with $\delta_{\text{initial}_i} = \text{cid}$ and $\rho_{\text{initial}_i} = \text{abort}$ ensures that:

1. The state token of currency cid is getting burned $\{\text{ST} \mapsto -1\} \subseteq \text{mint}$.

Each spent ν_{commit} validator with $\delta_{\text{commit}_i} = (\text{cid}, \cdot)$ and $\rho_{\text{commit}_i} = \text{abort}$ ensures that:

1. The state token of currency cid is getting burned $\{\text{ST} \mapsto -1\} \subseteq \text{mint}$.

The $\mu_{\text{head}}(\phi_{\text{seed}})$ minting policy governs the burning of tokens via redeemer burn that:

1. All tokens in mint need to be of negative quantity $\forall \{\text{cid} \mapsto \cdot \mapsto q\} \in \text{mint} : q < 0$.

[4]Only the first m outputs are used for reimbursing, while more outputs may be present in the transaction, e.g for returning change.

5.5 Close Transaction

In order to close a head, a head member may post the *close* transaction (see Figure 7). This transaction has

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head}.

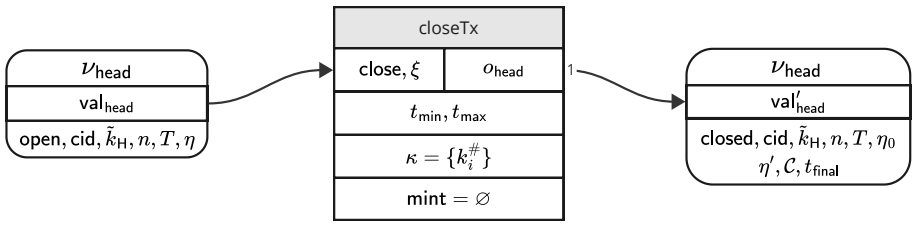

Figure 7: *close* transaction spending the open head output and producing a closed head output.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{close}, \xi)$, where ξ is a multi-signature of the to be closed snapshot, and checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{open}$ to $\delta'_{\mathsf{head}} \sim \mathsf{closed}$, parameters $\mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T$ stay unchanged and the new state is governed again by ν_{head}:

$$(\mathsf{open}, \mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T, \eta) \xrightarrow[\xi]{\mathsf{close}} (\mathsf{closed}, \mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T, \eta_0, \eta', \mathcal{C}, t_{\mathsf{final}})$$

2. Records the initial snapshot state $\eta_0 = \eta$.
   This makes off-chain signatures rollback and replay resistant, see 6.5 for details.
3. New snapshot state is the initial η_0 or correctly signed by all participants in ξ.
   Given the closed snapshot number s' from $(s', \cdot) = \eta'$,

$$\begin{cases} \mathsf{MS\text{-}Verify}(\tilde{k}_{\mathsf{H}}, \mathsf{msg}, \xi) = \mathtt{true} & \text{if } s' > 0, \\ \eta' = \eta_0 & \text{otherwise.} \end{cases}$$

   where msg is the concatenated bytes of cid, η_0 and η': $\mathsf{msg} = (\mathsf{cid}||\eta_0||\eta')$.
4. Initializes the set of contesters as $\mathcal{C} = \emptyset$.
   This allows the closing party to also contest and is required for use cases where pre-signed, valid in the future, close transactions are used to delegate head closing.
5. Correct contestation deadline is set $t_{\mathsf{final}} = t_{\mathsf{max}} + T$.
6. Transaction validity range is bounded by $t_{\mathsf{max}} - t_{\mathsf{min}} \leq T$.
   This ensures the contestation deadline t_{final} is at most $2 * T$ in the future.
7. Value in the head is preserved $\mathsf{val}'_{\mathsf{head}} = \mathsf{val}_{\mathsf{head}}$.
8. Transaction is signed by a participant $\exists\{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$.
9. No minting or burning $\mathsf{mint} = \varnothing$.

5.6 Contest Transaction

The *contest* transaction (see Figure 8) is posted by a party to prove the currently closed state is not the latest one. This transaction has

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head}.

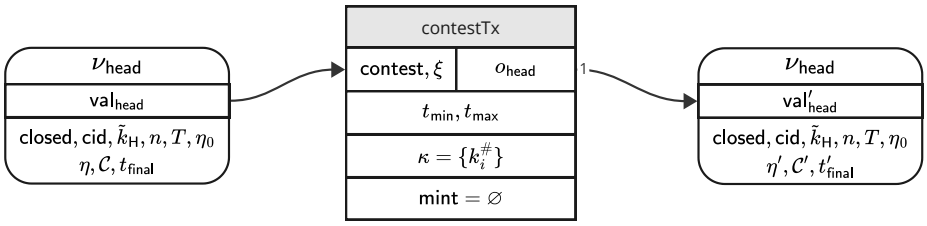

Figure 8: *contest* transaction spending the closed head output and producing a different closed head output.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{contest}, \xi)$, where ξ is a multi-signature of the contest snapshot, and checks:

1. State stays closed in both δ_{head} and δ'_{head}, parameters $\mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T, \eta_0$ stay unchanged and the new state is governed again by ν_{head}:

$$(\mathsf{closed}, \mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T, \eta_0, \eta, \mathcal{C}, t_{\mathsf{final}}) \xrightarrow[\xi]{\mathsf{contest}} (\mathsf{closed}, \mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T, \eta_0, \eta', \mathcal{C}', t'_{\mathsf{final}})$$

2. Contest snapshot is newer $s' > s$, where $(s, \cdot) = \eta$ is the current and $(s', \cdot) = \eta'$ is the contest snapshot number.
3. ξ is a valid multi-signature of the new snapshot state $\mathsf{MS\text{-}Verify}(\tilde{k}_{\mathsf{H}}, (\mathsf{cid}||\eta_0||\eta'), \xi) = \texttt{true}$.
4. The single signer $\{k^{\#}\} = \kappa$ has not already contested $k^{\#} \notin \mathcal{C}$ and is added to the set of contesters $\mathcal{C}' = \mathcal{C} \cup k^{\#}$.
5. Transaction is posted before deadline $t_{\mathsf{max}} \leq t_{\mathsf{final}}$.
6. Contestation deadline is updated correctly to

$$t'_{\mathsf{final}} = \begin{cases} t_{\mathsf{final}} & \text{if } |\mathcal{C}'| = n, \\ t_{\mathsf{final}} + T & \text{otherwise.} \end{cases}$$

7. Transaction is signed by a participant $\exists\{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$.
8. Value in the head is preserved $\mathsf{val}'_{\mathsf{head}} = \mathsf{val}_{\mathsf{head}}$.
9. No minting or burning $\mathsf{mint} = \varnothing$.

5.7 Fan-Out Transaction

Once the contestation phase is over, a head may be finalized by posting a *fanout* transaction (see Figure 9), which distributes UTxOs from the head according to the latest state. It consists of

- one input spending from ν_{head} holding the ST, and
- outputs $o_1 \dots o_m$ to distribute UTxOs.

Note that *fanout* represents a final transition of the state machine and hence there is no state machine output.

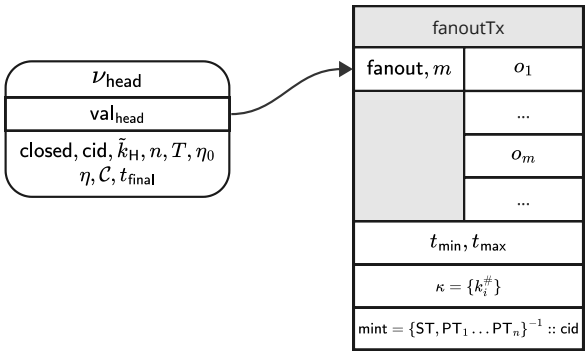

Figure 9: *fanout* transaction spending the closed head output and distributing funds with outputs $o_1 \dots o_m$.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{fanout}, m)$, where m is the number of outputs to distribute, and checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{closed}$ to terminal state final:

$$(\mathsf{closed}, \mathsf{cid}, \tilde{k}_{\mathsf{H}}, n, T, \eta_0, \eta, \mathcal{C}, t_{\mathsf{final}}) \xrightarrow[m]{\mathsf{fanout}} \mathsf{final}$$

2. The first m outputs are distributing funds according to $(\cdot, U^{\#}) = \eta$. That is, the outputs exactly correspond to the UTxO canonically combined $U^{\#}$ (see Section 5.3):

$$\mathsf{hash}(\bigoplus_{j=1}^{m} \mathsf{bytes}(o_j)) = U^{\#}$$

3. Transaction is posted after contestation deadline $t_{\mathsf{min}} > t_{\mathsf{final}}$.
4. All tokens are burnt $|\{\mathsf{cid} \mapsto \cdot \mapsto -1\} \in \mathsf{mint}| = n + 1$.

The $\mu_{\mathsf{head}}(\phi_{\mathsf{seed}})$ minting policy governs the burning of tokens via redeemer burn that:

1. All tokens in mint need to be of negative quantity $\forall\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint} : q < 0$.

6 Off-Chain Protocol

This section describes the actual Coordinated Hydra Head protocol, an even more simplified version of the original publication [8]. See the protocol overview in Section 2 for an introduction and notable changes to the original protocol. While the on-chain part already describes the full life-cycle of a Hydra head on-chain, this section completes the picture by defining how the protocol behaves off-chain and notably the relationship between on- and off-chain semantics. Participants of the protocol are also called Hydra head members, parties or simply protocol actors. The protocol is specified as a reactive system that processes three kinds of events:

1. On-chain protocol transactions as introduced in Section 5, which are posted to the mainchain and can be observed by all actors
2. Off-chain network messages sent between protocol actors (parties):
   - `reqTx`: to request a transaction to be included in the next snapshot
   - `reqSn`: to request a snapshot to be created & signed by every head member
   - `ackSn`: to acknowledge a snapshot by replying with their signatures
3. Commands issued by the participants themselves or on behalf of end-users and clients
   - `init`: to start initialization of a head
   - `newTx`: to submit a new transaction to an open head
   - `close`: to request closure of an open head

The behavior is fully specified in Figure 10, while the following paragraphs introduce notation, explain variables and walk-through the protocol flow.

6.1 Assumptions

On top of the statements of the protocol setup in Section 4, the off-chain protocol logic relies on these assumptions:

move/merge with protocol setup?

- Every network message received from a specific party is checked for authentication. An implementation of the specification needs to find a suitable means of authentication, either on the communication channel or for individual messages. Unauthenticated messages must be dropped.
- The head protocol gets correctly (and with completeness) notified about observed transactions on-chain belonging to the respective head instance.
- All events are processed to completion, i.e. run-to-completion semantics and no preemption.
- Events are deduplicated. That is, any two identical events must not lead to multiple invocations of the handling semantics.
- Given the specification, events may pile up forever and implementations need to consider these situations (i.e. potential for DoS). A valid reaction to this would be to just drop these events. Note that, from a security standpoint, these situations are identical to a non-collaborative peer and closing the head is also a possible reaction.

- The lifecycle of a Hydra head on-chain does not cross (hard fork) protocol update boundaries. Note that these events are announced in advance hence it should be possible for implementations to react in such a way as to expedite closing of the head before such a protocol update. This further assumes that the contestation period parameter is picked accordingly.

Treat this also in a dedicated section like roll-backs

6.2 Notation

- **on** *event* specifies how the protocol reacts on a given *event*. Further information may be available from the constituents of *event* and origin of the event.
- **require** p means that boolean expression $p \in \mathbb{B}$ must be satisfied for the further execution of a routine, while discontinued on $\neg p$. A conservative protocol actor could interpret this as a reason to close the head.

missing:, apply tx

- **wait** p is a non-blocking wait for boolean predicate $p \in \mathbb{B}$ to be satisfied. On $\neg p$, the execution of the routine is stopped, queued, and reactivated at latest when p is satisfied.
- **multicast** msg means that a message msg is (channel-) authenticated and sent to all participants of this head, including the sender.
- **postTx** tx has a party create transaction tx, potentially from some data, and submit it on-chain. See Section 5 for individual transaction details.
- **output** *event* signals an observation of *event*, which is used in the security definition and proofs of Section 7. This keyword can be ignored when implementing the protocol.

6.3 Variables

Besides parameters agreed in the protocol setup (see Section 4), a party's local state consists of the following variables:

- $\hat{s}$: Sequence number of latest seen snapshot.
- $\bar{s}$: Sequence number of latest confirmed snapshot.
- $\bar{\sigma}$: Signature associated with the latest confirmed snapshot.
- $\hat{\mathcal{U}}$: UTxO set of the latest seen snapshot.
- $\bar{\mathcal{U}}$: UTxO set associated with the latest confirmed snapshot.
- $\hat{\Sigma} \in (\mathbb{N} \times \mathbb{H})^*$: Accumulator of signatures of the latest seen snapshot, indexed by parties.
- $\hat{\mathcal{L}}$: UTxO set representing the local ledger state resulting from applying $\hat{\mathcal{T}}$ to $\bar{\mathcal{U}}$ to validate requested transactions.
- $\hat{\mathcal{T}} \in \mathcal{T}^*$: List of transactions applied locally and pending inclusion in a snapshot (if this party is the next leader).
- $\mathcal{T}_{\mathsf{all}} \in (\mathbb{H} \times \mathcal{T})^*$: Associative list of all seen transactions not yet included in a snapshot.

6.4 Protocol flow

Make consistent with figure again

6.4.1 Initializing the head

`init`. Before a head can be initialized, all parties need to exchange and agree on protocol parameters during the protocol setup phase (see Section 4), so we can assume the public Cardano keys $\underline{k}_{\mathsf{C}}^{setup}$, Hydra keys $\tilde{k}_{\mathsf{H}}^{setup}$, as well as the contestation period T^{setup} are available. One of the clients then can start head initialization using the `init` command, which will result in an *init* transaction being posted.

`initialTx`. All parties will receive this *init* transaction and validate announced parameters against the pre-agreed *setup* parameters, as well as the structure of the transaction and the minting policy used. This is a vital step to ensure the initialized Head is valid, which cannot be checked completely on-chain (see also Section 5.1).

`commitTx`. As each party p_j posts a *commit* transaction, the protocol records observed committed UTxOs of each party C_j. With all committed UTxOs known, the η-state is created (as defined in Section 5.3) and the *collectCom* transaction is posted. Note that while each participant may post this transaction, only one of them will be included in the blockchain as the mainchain ledger prevents double spending. Should any party want to abort, they would post an *abort* transaction and the protocol would end at this point.

`collectComTx`. Upon observing the *collectCom* transaction, the parties compute $U_0 \leftarrow \bigcup_{j=1}^{n} C_j$ using previously observed C_j and initialize $\hat{\mathcal{U}} = \bar{\mathcal{U}} = \hat{\mathcal{L}} = U_0$ with it. The initial transaction sets are empty $\mathcal{T} = \bar{\mathcal{T}} = \hat{\mathcal{T}} = \emptyset$, and $\bar{s} = \hat{s} = 0$.

6.4.2 Processing transactions off-chain

Transactions are announced and captured in so-called snapshots. Parties generate snapshots in a strictly sequential round-robin manner. The party responsible for issuing the i^{th} snapshot is the *leader* of the i^{th} snapshot. Leader selection is round-robin per the $\underline{k}_{\mathsf{H}}$ from the protocol setup. While the frequency of snapshots in the general Head protocol [8] was configurable, the Coordinated Head protocol does specify a snapshot to be created after each transaction.

`newTx`. At any time, by sending request $(\texttt{newTx}, \text{tx})$, a client of the protocol can submit a new transaction tx to the head, which results in it being sent out to all parties as a $(\texttt{reqTx}, \text{tx})$ message.

`reqTx`. Upon receiving request $(\texttt{reqTx}, \text{tx})$, the transaction gets recorded in $\mathcal{T}_{\mathsf{all}}$ and applied to the *local* ledger state $\hat{\mathcal{L}} \circ \text{tx}$. If not applicable yet, the protocol does **wait** to retry later or eventually marks this transaction as invalid (see assumption about events piling up). After applying and if there is no current snapshot "in flight" ($\hat{s} = \bar{s}$) and the receiving party p_i is the next snapshot leader, a message to request snapshot signatures `reqSn` is sent.

reqSn. Upon receiving request $(\texttt{reqSn}, s, \mathcal{T}^{\#}_{\mathsf{req}})$ from party p_j, the receiver p_i checks that s is the next snapshot number and that party p_j is responsible for leading its creation. Party p_i has to **wait** until the previous snapshot is confirmed ($\bar{s} = \hat{s}$) and all requested transaction hashes $\mathcal{T}^{\#}_{\mathsf{req}}$ can be resolved in $\mathcal{T}_{\mathsf{all}}$. Then, all those resolved transactions $\mathcal{T}_{\mathsf{req}}$ are **require**d to be applicable to $\bar{\mathcal{U}}$, otherwise the snapshot is rejected as invalid. Only then, p_i increments their seen-snapshot counter $\hat{s}$, resets the signature accumulator $\hat{\Sigma}$, and computes the UTxO set $\hat{\mathcal{U}}$ of the new (seen) snapshot as $\hat{\mathcal{U}} \leftarrow \bar{\mathcal{U}} \circ \mathcal{T}_{\mathsf{req}}$. Then, p_i creates a signature σ_i using their signing key k^{sig}_{H} on a message comprised by the cid, the η_0 corresponding to the initial UTxO set U_0, and the new η' given by the new snapshot number $\hat{s}$ and canonically combining $\hat{\mathcal{U}}$ (see Section 5.5 for details). The signature is sent to all head members via message $(\texttt{ackSn}, \hat{s}, \sigma_i)$. Finally, the pending transaction set $\hat{\mathcal{T}}$ gets pruned by re-applying all locally pending transactions $\hat{\mathcal{T}}$ to the just requested snapshot's UTxO set $\hat{\mathcal{U}}$ iteratively and ultimately yielding a "pruned" version of $\hat{\mathcal{T}}$ and $\hat{\mathcal{U}}$. Also, the set of all transactions $\mathcal{T}_{\mathsf{all}}$ can be reduced by the requested transactions $\mathcal{T}_{\mathsf{req}}$.

define leader

ackSn. Upon receiving acknowledgment $(\texttt{ackSn}, s, \sigma_j)$, all participants **require** that it is from an expected snapshot (either the last seen $\hat{s}$ or $+\ 1$), potentially **wait** for the corresponding `reqSn` such that $\hat{s} = s$ and **require** that the signature is not yet included in $\hat{\Sigma}$. They store the received signature in the signature accumulator $\hat{\Sigma}$, and if the signature from each party has been collected, p_i aggregates the multisignature $\tilde{\sigma}$ and **require** it to be valid. If everything is fine, the snapshot can be considered confirmed by updating $\bar{s} = s$ and participants also store the UTxO set in $\bar{\mathcal{U}}$, as well as the signature in $\bar{\sigma}$ for later reference. Similar to the `reqTx`, if p_i is the next snapshot leader and there are already transactions to snapshot in $\hat{\mathcal{T}}$, a corresponding `reqSn` is distributed.

6.4.3 Closing the head

close. In order to close a head, a client issues the `close` event which uses the latest confirmed snapshot $\bar{\mathcal{U}}$ to create

- the new η-state η' from the last confirmed UTxO set and snapshot number, and
- the certifiate ξ using the corresponding multi-signature.

With η' and ξ, the *close* transaction can be constructed and posted. See Section 5.5 for details about this transaction.

closeTx/contestTx. When a party observes the head getting closed or contested, the η-state extracted from the *close* or *contest* transaction represents the latest head status that has been aggregated on-chain so far (by a sequence of *close* and *contest* transactions). If the last confirmed (off-chain) snapshot is newer than the observed (on-chain) snapshot number s_c, an updated η-state and certificate ξ is constructed posted in a *contest* transaction (see Section 5.6).

6.5 Rollbacks and protocol changes

The overall life-cycle of the Head protocol is driven by on-chain events (see introduction of Section 6) which stem from observing transactions on the mainchain. Most blockchains, however, do only provide *eventual* consistency. The consensus algorithm ensures a consistent view of the

Discuss protocol updates as well, also in light of rollbacks

history of blocks and transactions between all parties, but this so-called *finality* is only achieved after some time and the local view of the blockchain history may change until that point.

On Cardano with it's Ouroboros consensus algorithm, this means that any local view of the mainchain may not be the longest chain and a node may switch to a longer chain, onto another fork. This other version of the history may not include what was previously observed and hence, any tracking state needs to be updated to this "new reality". Practically, this means that an observer of the blockchain sees a *rollback* followed by rollforwards.

For the Head protocol, this means that chain events like `closeTx` may be observed a second time. Hence, it is crucial, that the local state of the Hydra protocol is kept in sync and also rolled back accordingly to be able to observe and react to these events the right way, e.g. correctly contesting this `closeTx` if need be.

The rollback handling can be specified fully orthogonal on top of the nominal protocol behavior, if the chain provides strictly monotonically increasing points p on each chain event via a new or wrapped `rollforward` event and `rollback` event with the point to which a rollback happened:

`rollforward`. On every chain event that is paired or wrapped in a rollforward event $(\texttt{rollback}, p)$ with point p, protocol participants store their head state indexed by this point in a history Ω of states $\Delta \leftarrow (\hat{s}, \bar{s}, \bar{\sigma}, \hat{\mathcal{U}}, \bar{\mathcal{U}}, \hat{\Sigma}, \hat{\mathcal{L}}, \mathcal{T}_{all}, \hat{\mathcal{T}})$ and $\Omega' = (p, \Delta) \cup \Omega$.

`rollback`. On a rollback $(\texttt{rollback}, p_{rb})$ to point p_{rb}, the corresponding head state Δ need to be retrieved from Ω, with the maximal point $p \leq p_{rb}$, and all entries in Ω with $p > p_{rb}$ get removed.

This will essentially reset the local head state to the right point and allow the protocol to progress through the life-cycle normally. Most stages of the life-cycle are unproblematic if they are rolled back, as long as the protocol logic behaves as in the nominal case.

A rollback "past open" is a special situation though. When a Head is open and snapshots have been signed, but then a *collectCom* and one or more *commit* transactions were rolled back, a bad actor could choose to commit a different UTxO and open the Head with a different initial UTxO set, while the already signed snapshots would still be (cryptographically) valid. To mitigate this, all signatures on snapshots need to incorporate the initial UTxO set by including η_0.

Write about contestation deadline vs. rollbacks

In figure: combine on UTxO slightly different than on commits

Coordinated Hydra Head

on $(\texttt{init})$ *from client*
- $n \leftarrow |\underline{k}_{\mathsf{H}}^{setup}|$
- $\tilde{k}_{\mathsf{H}} \leftarrow \mathsf{MS\text{-}AVK}(\underline{k}_{\mathsf{H}}^{setup})$
- $\underline{k}_{\mathsf{C}} \leftarrow \underline{k}_{\mathsf{C}}^{setup}$
- $T \leftarrow T^{setup}$
- **postTx** $(init, n, \tilde{k}_{\mathsf{H}}, \underline{k}_{\mathsf{C}}, T)$

on $(\texttt{initialTx}, \mathsf{cid}, \phi_{\mathsf{seed}}, n, \tilde{k}_{\mathsf{H}}, \underline{k}_{\mathsf{C}}^{\#}, T)$ *from chain*
- **require** $\tilde{k}_{\mathsf{H}} = \mathsf{MS\text{-}AVK}(\underline{k}_{\mathsf{H}}^{setup})$
- **require** $\underline{k}_{\mathsf{C}}^{\#} = [\mathsf{hash}(k) \mid \forall k \in \underline{k}_{\mathsf{C}}^{setup}]$
- **require** $T = T^{setup}$
- **require** $\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\phi_{\mathsf{seed}}))$

on $(\texttt{commitTx}, j, U)$ *from chain*
- $C_j \leftarrow U$
- **if** $\forall k \in [1..n] : C_k \neq \text{undef}$
  - $\eta \leftarrow (0, \mathsf{combine}([C_1 \dots C_n]))$
  - **postTx** $(collectCom, \eta)$

on $(\texttt{collectComTx}, \eta_0)$ *from chain*
- $U_0 \leftarrow \bigcup_{j=1}^{n} U_j$
- $\hat{\mathcal{U}}, \bar{\mathcal{U}}, \hat{\mathcal{L}} \leftarrow U_0$
- $\hat{s}, \bar{s} \leftarrow 0$
- $\mathcal{T}, \hat{\mathcal{T}}, \bar{\mathcal{T}} \leftarrow \emptyset$

---

on $(\texttt{newTx}, \text{tx})$ *from client*
- **multicast** $(\texttt{reqTx}, \text{tx})$

on $(\texttt{reqTx}, \text{tx})$ *from* p_j
- $\mathcal{T}_{\mathsf{all}} \leftarrow \mathcal{T}_{\mathsf{all}} \cup \{(\mathsf{hash}(\text{tx}), \text{tx})\}$
- **wait** $\hat{\mathcal{L}} \circ \text{tx} \neq \perp$
  - $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} \circ \text{tx}$
  - $\hat{\mathcal{T}} \leftarrow \hat{\mathcal{T}} \cup \{\text{tx}\}$
  - **if** $\hat{s} = \bar{s} \wedge \mathsf{leader}(\bar{s}+1) = i$
    - **multicast** $(\texttt{reqSn}, \bar{s}+1, \hat{\mathcal{T}})$

on $(\texttt{reqSn}, s, \mathcal{T}_{req}^{\#})$ *from* p_j
- **require** $s = \hat{s}+1 \ \wedge \ \mathsf{leader}(s) = j$
- **wait** $\bar{s} = \hat{s} \ \wedge \ \forall h \in \mathcal{T}_{req}^{\#} : (h, \cdot) \in \mathcal{T}_{\mathsf{all}}$
  - $\mathcal{T}_{\mathsf{req}} \leftarrow \{\mathcal{T}_{\mathsf{all}}[h] \mid \forall h \in \mathcal{T}_{req}^{\#}\}$
  - **require** $\bar{\mathcal{U}} \circ \mathcal{T}_{\mathsf{req}} \neq \perp$
  - $\hat{\mathcal{U}} \leftarrow \bar{\mathcal{U}} \circ \mathcal{T}_{\mathsf{req}}$
  - $\hat{s} \leftarrow \bar{s}+1$
  - $\eta' \leftarrow (\hat{s}, \mathsf{combine}(\hat{\mathcal{U}}))$
  - $\sigma_i \leftarrow \mathsf{MS\text{-}Sign}(k_{\mathsf{H}}^{sig}, (\mathsf{cid}||\eta_0||\eta'))$
  - $\hat{\Sigma} \leftarrow \emptyset$
  - **multicast** $(\texttt{ackSn}, \hat{s}, \sigma_i)$
  - $\forall \text{tx} \in \mathcal{T}_{\mathsf{req}}$: **output** $(\texttt{seen}, \text{tx})$
  - $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{U}}$
  - $X \leftarrow \hat{\mathcal{T}}$
  - $\hat{\mathcal{T}} \leftarrow \emptyset$
  - **for** $\text{tx} \in X : \hat{\mathcal{L}} \circ \text{tx} \neq \perp$
    - $\hat{\mathcal{T}} \leftarrow \hat{\mathcal{T}} \cup \{\text{tx}\}$ $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} \circ \text{tx}$
  - $\mathcal{T}_{\mathsf{all}} \leftarrow \{tx \mid \forall tx \in \mathcal{T}_{\mathsf{all}} : tx \notin \mathcal{T}_{\mathsf{req}}\}$

on $(\texttt{ackSn}, s, \sigma_j)$ *from* p_j
- **require** $s \in \{\hat{s}, \hat{s}+1\}$
- **wait** $\hat{s} = s$
  - **require** $(j, \cdot) \notin \hat{\Sigma}$
  - **if** $\forall k \in [1..n] : (k, \cdot) \in \hat{\Sigma}$
    - $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\underline{k}_{\mathsf{H}}^{setup}, \hat{\Sigma})$
    - $\eta' \leftarrow (\hat{s}, \mathsf{combine}(\hat{\mathcal{U}}))$
    - **require** $\mathsf{MS\text{-}Verify}(\tilde{k}_{\mathsf{H}}, (\mathsf{cid}||\eta_0||\eta'), \tilde{\sigma})$
    - $\bar{\mathcal{U}} \leftarrow \hat{\mathcal{U}}$
    - $\bar{s} \leftarrow \hat{s}$
    - $\bar{\sigma} \leftarrow \tilde{\sigma}$
    - $\forall \text{tx} \in \mathcal{T}_{\mathsf{req}}$: **output** $(\texttt{conf}, \text{tx})$
    - **if** $\mathsf{leader}(s+1) = i \wedge \hat{\mathcal{T}} \neq \emptyset$
      - **multicast** $(\texttt{reqSn}, s+1, \hat{\mathcal{T}})$

---

on $(\texttt{close})$ *from client*
- $\eta' \leftarrow (\bar{s}, \mathsf{combine}(\bar{\mathcal{U}}))$
- $\xi \leftarrow \bar{\sigma}$
- **postTx** $(close, \eta', \xi)$

on $(\texttt{closeTx}, \eta) \vee (\texttt{contestTx}, \eta)$ *from chain*
- $(s_c, \cdot) \leftarrow \eta$
- **if** $\bar{s} > s_c$
  - $\eta' \leftarrow (\bar{s}, \mathsf{combine}(\bar{\mathcal{U}}))$
  - $\xi \leftarrow \bar{\sigma}$
  - **postTx** $(contest, \eta', \xi)$

Figure 10: Head-protocol machine for the *coordinated head* from the perspective of party p_i.

7 Security (WIP — Iteration 1)

The security analysis is still **sketchy**, with the goal to make it more formal in upcoming iterations

Add security experiment

above this section there is no security game

Adversaries:

Active Adversary. An *active adversary* $\mathcal{A}$ has full control over the protocol, i.e., he is fully unrestricted in the above security game.

Network Adversary. A *network adversary* $\mathcal{A}_\emptyset$ does not corrupt any head parties, eventually delivers all sent network messages (i.e., does not drop any messages), and does not cause the `close` event. Apart from this restriction, the adversary can act arbitrarily in the above experiment.

Random variables:

- $\hat{S}_i$: the set of transactions tx for which party p_i, *while uncorrupted*, output $(\texttt{seen}, \text{tx})$;
- $\bar{C}_i$: the set of transactions tx for which party p_i, *while uncorrupted*, output $(\texttt{conf}, \text{tx})$;
- $\bar{\Sigma}_i$: latest snapshot (s, U) that party p_i performed *while uncorrupted*: output $(\texttt{snap}, (s, U))$;
- H_{cont}: the set of (at the time) uncorrupted parties who produced ξ upon close/contest request and ξ was applied to correct η; and
- $\mathcal{H}$: the set of parties that remain uncorrupted.

Security conditions / events:

- Consistency (Head): In presence of an active adversary, the following condition holds at any point in time: For all i, j, $U_0 \circ (\bar{C}_i \cup \bar{C}_j) \neq \bot$, i.e., no two uncorrupted parties see conflicting transactions confirmed.
- Oblivious Liveness (Head): Consider any protocol execution in presence of a network adversary wherein the head does not get closed for a sufficiently long period of time, and consider an honest party p_i who enters transaction tx by executing $(\texttt{newTx}, \text{tx})$ *each time after having finished a snapshot.*

  Then the following eventually holds: $\text{tx} \in \bigcap_{i\in[n]} \bar{C}_i \ \vee \ \forall i : U_0 \circ (\bar{C}_i \cup \{\text{tx}\}) = \bot$, i.e., every party will observe the transaction confirmed or every party will observe the transaction in conflict with their confirmed transactions.[5]
- Soundness (Chain): In presence of an active adversary, the following condition is satisfied: $\exists \tilde{S} \subseteq \bigcap_{i\in\mathcal{H}} \hat{S}_i : U_{\mathsf{final}} = U_0 \circ \tilde{S} \neq \bot$, i.e., the final UTxO set results from applying a set of transactions to U_0 that have been seen by all honest parties (wheras each such transaction applies conforming to the ledger rules).
- Completeness (Chain): In presence of an active adversary, the following condition holds: For $\tilde{S}$ as above, $\bigcup_{p_i\in H_{\mathsf{cont}}} \bar{C}_i \subseteq \tilde{S}$, i.e., all transactions seen as confirmed by an honest party at the end of the protocol are considered.

[5] In particular, *liveness* expresses that the protocol makes progress under reasonable network conditions if no head parties get corrupted.

Note that the original version of the coordinated head satisfies a stronger version of liveness which is important for the 'user experience' in the protocol:

- Liveness (Head): Consider any protocol execution in presence of a network adversary wherein the head does not get closed for a sufficiently long period of time, and consider an honest party p_i who enters transaction tx by executing $(\texttt{newTx}, \text{tx})$.

  Then the following eventually holds: $\text{tx} \in \bigcap_{i\in[n]} \bar{C}_i \ \vee \ \forall i : U_0 \circ (\bar{C}_i \cup \{\text{tx}\}) = \bot$, i.e., every party will observe the transaction confirmed or every party will observe the transaction in conflict with their confirmed transactions.[6]

7.1 Proofs

Consistency.

Lemma 1 (Consistency). *The coordinated head protocol satisfies the* Consistency *property.*

Proof. Observe that $\bar{C}_i \cup \bar{C}_j \subseteq \hat{S}_i$ since no transaction can be confirmed without every honest party signing off on it. Since parties do not sign conflicting transactions (see `reqSn`, 'wait'), we have $U_0 \circ \bar{C}_i \neq \bot$, $U_0 \circ \bar{C}_j \neq \bot$, and $U_0 \circ \hat{S}_i \neq \bot$. Thus, since $\bar{C}_i \cup \bar{C}_j \subseteq \hat{S}_i$ it follows that $U_0 \circ (\bar{C}_i \cup \bar{C}_j) \neq \bot$ □

Oblivious Liveness. For all lemmas towards oblivious liveness, we assume the presence of a network adversary, and that the head does not get closed for a sufficiently long period of time. We call this the *liveness condition.*

Lemma 2. *Under the liveness condition, any snapshot issued as* $(\texttt{reqSn}, s, T)$ *will eventually be confirmed in the sense that every party holds a valid mulisignature on it.*

Proof. Consider a party p_i receiving message $(\texttt{reqSn}, s, T)$. We demonstrate that p_i executes the code past the 'wait' instruction of the `reqSn` routine.

- Passing the 'require' guard: Note that the snapshot leader sends the request only if $\hat{s} = \bar{s}$, and for $s = \hat{s} + 1$. Thus, $\hat{s}_i = \hat{s}$ since p_i has already signed the snapshot for $\hat{s}$. The 'require' guard is thus satisfied for p_i.
- Passing the 'wait' guard: Since the snapshot leader sees $\hat{s} = \bar{s}$, also p_i will eventually see $\hat{s}_i = \bar{s}_i$. Furthermore, since all leaders are honest, it holds that $\hat{\mathcal{U}} \circ \mathcal{T}_{res} \neq \bot$ by construction.

This implies that every party will eventually sign and acknowledge the newly created snapshot. Finally, the 'require' and 'wait' guards of the `ackSn` code will be passed by every party since an `ackSn` for snapshot number s can only be received for $s \in \{\hat{s}, \hat{s}+1\}$ as an acknowledgement can only be received for the current snapshot being worked on by p_i or a snapshot that is one step ahead—implying that everybody will hold a valid multisignature on the snapshot in consideration. □

Lemma 3 (Eternal snapshot confirmation). *Under the liveness condition, as long as new transactions are issued, for any* $k > 0$*, every party eventually confirms a snapshot with sequence number* $s = k$*.*

[6] In particular, *liveness* expresses that the protocol makes progress under reasonable network conditions if no head parties get corrupted.

Proof. By Lemma 2, any requested snapshot eventually gets confirmed, implying that the next leader observes $\hat{s} = \bar{s}$ and thus, in turn, issues a new snapshot. Thus, for any k, a snapshot is eventually confirmed. □

Lemma 4 (Oblivious Liveness). *The coordinated head protocol satisfies the* Oblivious Liveness *property.*

Proof. Consider the first point in time where a transaction tx enters the system by some party p_i issuing $(\texttt{newTx}, \text{tx})$, and consider the next point in time t when p_i issues a snapshot.

By Lemma 3, this snapshot will eventually be issued and confirmed by all parties.

Let $\hat{\mathcal{T}}$ be the transactions to be considered by p_i's snapshot: $\hat{\mathcal{L}} = \bar{\mathcal{U}} \circ \hat{\mathcal{T}}$ where $\bar{\mathcal{U}}$ is the snapshot prior to p_i's. Since p_i issues $(\texttt{reqTx}, \text{tx})$ after each snapshot, we have that, either,

- $\text{tx} \in \hat{\mathcal{T}}$, in which case $\text{tx} \in \bigcap_{i \in [n]} \bar{C}_i$ after everybody has completed this snapshot, or,
- $\text{tx} \notin \hat{\mathcal{T}}$, in which case $\hat{\mathcal{L}} \circ \text{tx} = \bot$ (tx is still in the wait queue of $(\texttt{reqTx}, \text{tx})$. After everybody has completed this snapshot, it thus holds that $\forall i : U_0 \circ \bar{C}_i = \hat{\mathcal{L}}$, and thus, that $\forall i : U_0 \circ (\bar{C}_i \cup \{\text{tx}\}) = \bot$.

In both cases, the lemma follows. □

Soundness and completeness.

Lemma 5 (Soundness). *The basic head protocol satisfies the* Soundness *property.*

Proof. Let T be the set of transactions such that $U_{\mathsf{final}} = U_0 \circ T$. Since U_{final} is multi-signed, it holds that $T \subseteq \hat{S}_i$ (T is *seen*) by every honest party in the head. Furthermore, since honest signatures are only issued for valid transaction, $U_{\mathsf{final}} \neq \bot$ (i.e., U_{final} is a valid state), and soundness follows. □

Lemma 6 (Completeness). *The basic head protocol satisfies the* Completeness *property.*

Proof. Consider all parties $p_i \in H_{\mathsf{cont}}$. Since the close/contest process finally accepts the latest multi-signed snapshot, it holds that $U_{\mathsf{final}}.s \geq \max_{p_i \in H_{\mathsf{cont}}}(\bar{s}_i)$, and thus that $\bigcup_{p_i \in H_{\mathsf{cont}}} \bar{C}_i \subseteq \bigcap_{p_i \in \mathcal{H}} \hat{S}_i$, and completeness follows. □

References

[1] Extended UTXO-2 model. https://github.com/hydra-supplementary-material/eutxo-spec/blob/master/extended-utxo-specification.pdf.

[2] A formal specification of the cardano ledger. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-ledger.pdf.

[3] A formal specification of the cardano ledger integrating plutus core. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/alonzo-ledger.pdf.

[4] Hydra repository. https://github.com/input-output-hk/hydra.

[5] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of Bitcoin transactions. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, pages 541–560, 2018.

[6] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended UTxO model. In *4th Workshop on Trusted Smart Contracts*, 2020. http://fc20.ifca.ai/wtsc/WTSC2020/WTSC20_paper_25.pdf.

[7] Manuel M. T. Chakravarty, James Chapman, Kenneth M. Mackenzie, Orestis Melkonian, Jann, Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, Joachim, and Zahnentferner. Utxoma: Utxo with multi-asset support. 2020.

[8] Manuel MT Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Hydra: Fast isomorphic state channels. *Cryptology ePrint Archive*, 2020.

[9] Kazuharu Itakura and Katsuhiro Nakamura. A public-key cryptosystem suitable for digital multisignatures. *NEC Research & Development*, (71):1–8, 1983.

[10] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: Extended abstract. pages 245–254, 2001.

[11] Joachim Zahnentferner. An abstract model of UTxO-based cryptocurrencies with scripts. *IACR Cryptology ePrint Archive*, 2018:469, 2018.